Techniques for GPU-based Color Quantization

Matthias Trapp
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam, Germany
matthias.trapp@hpi.de

Sebastian Pasewaldt
Digital Masterpieces GmbH
Germany
sebastian.pasewaldt@digitalmasterpieces.com

Jürgen Döllner
Hasso Plattner Institute,
Digital Engineering Faculty,
University of Potsdam, Germany
juergen.doellner@hpi.de

ABSTRACT

This paper presents a GPU-based approach to color quantization by mapping of arbitrary color palettes to input images using Look-Up Tables (LUTs). For it, different types of LUTs, their GPU-based generation, representation, and respective mapping implementations are described and their run-time performance is evaluated and compared.

Keywords: color palettes, color quantization, look-up tables

1 INTRODUCTION

1.1 Motivation and Applications

In computer graphics, a color palette P denotes a finite list of colors $P = C_0, \ldots, C_n$, each associated with a color space $C_i \in XYZ \subset \mathbb{R}^3$, with n being the total number of colors within a palette. In hardware or software palettes, n is limited by color bit-depth or number of simultaneously available total colors.

There are multiple applications for a fast color palette mapping approach supported by a Graphics Processing Unit (GPU)-based implementation. Besides enabling non-uniform color reduction, it can be used in stylized rendering or image and video "retro" abstraction operations that mimicking old devices, such as consoles.

1.2 Problem Statement

Given an *indexed image* $I[0,w] \times [0,h] \rightarrow i = 0, \ldots, n$ and a color palette P, a respective color mapping of an output raster image $G[0,w] \times [0,h]$ can be formulated as follows:

$$G[x,y] = \rho(I[x,y],P) \qquad \rho(i,P) = C_i \qquad (1)$$

where the *color mapping* function ρ selects the i^{th} color based on the *color index* i from the palette and assigns it to the output pixel located at $[x,y]$. Such LUT mapping is compact and can be resolved in constant time $O(1)$. However, this approach cannot be used for an image or video frame that is represented by a number of colors such that $I[0,w] \times [0,h] \rightarrow C \in XYZ$. For these types of *high-color* or *true-color* images, the color mapping process can be formulated as follows:

$$G[x,y] = \tau(I[x,y],P) = \underset{0 \leq i \leq n}{\arg\min} \left(\delta(I[x,y],C_i)\right) \qquad (2)$$

where a distance function $\delta : C_i \times C_j \mapsto \mathbb{R}_+$ computes a distance value between two colors. Thus, the mapping process requires the computation of the minimum color distance for a given color C and all available palette colors in P. Therefore, the overall run-time complexity grows linear with the number n of total palette colors.

With respect to implementing such a non-uniform color quantization for interactive rendering purposes, an approach should adhere to the following requirements:

Real-time Performance (R1): To support color palette mappings for interactive image and video applications, the process should be real-time capable.

Fast Palette Interchange (R2): For flexibility in application, different color palettes should interchangeable easily and fast by requiring only minimal state changes during rendering [1].

Support True-Color Images (R3): The mapping approach should not rely on an indexed color image representation and should support images comprising 8 bit precision or more per color channel.

1.3 Approach and Contributions

To approach the requirements above, we evaluate techniques for mapping an arbitrary input color C_{in} to an output color $C_{out} \in P$ on a per-pixel basis by computing a *minimum color distance* in CIELAB (Lab) color space. By completely preprocessing the minimum color distance search into color LUTs, the mapping process can then be performed in constant time [5] for the sake of increased memory consumptions. Therefore, this paper makes the following contributions:

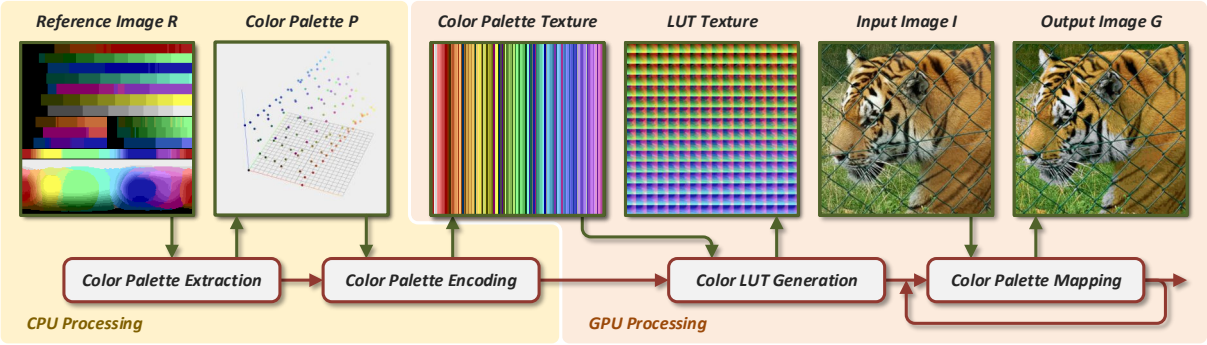


Figure 1: Overview of processing stages and data components as well as control (red) and data flow (green) of the presented real-time palette mapping approach by the example of color reduction using an Atari 800 color palette with 122 colors.

1. It presents approaches to fast color mapping of given color palettes to true-color input images in real-time using GPUs.

2. It provides respective implementation details based on Open Graphics Library (OpenGL) and evaluates the run-time performance.

The remainder of this paper is structured as follows. Section 2 describes the fundamental concept of our approach. Section 3 presents details of GPU-based implementations. Section 4 evaluates and discusses the run-time performance. Section 5 concludes this paper.

2 PALETTE MAPPING CONCEPT

Figure 1 shows an overview of the processing stages, components, and data structures of the GPU-based palette mapping approach. It basically comprises the following steps, covered in the remainder of this section in greater detail:

Color Palette Extraction: As a first (optional) step, a *color palette P* is extracted from a given input *reference image R* (Section 2.1).

Color Palette Encoding: To enable GPU-based processing of the subsequent steps, the extracted color palette P is encoded into a *color palette texture*.

Look-Up Table Generation: To enable an efficient shader-based color mapping implementation, the color palette texture is transformed into a *Look-Up Table (LUT)* (Section 2.3) based on a minimum color distance search (Section 2.2).

Color Palette Mapping: While the previous steps are only required to be performed once per color palette, the color palette mapping use the LUT to implement a point-based operation to map all pixel of multiple input images or video frames I to its respective output representations O using fragment or compute shader programs (Section 3.2).

2.1 Color Palette Extraction

There are basically two approaches for creating a color palette: (1) a user *explicitly* chooses a number of colors or (2) the color palette is *extracted* from a given reference image R.

To support the latter, a tool is created that extracts the color palette from a given image, which is required to be stored without lossy compression, in a preprocessing step. For it, the color of each pixel is added to a set of palette colors resulting in an unordered list of unique colors, which can be stored as image file. This process can be automated and batched for a number of input images. For our purposes, we obtain existing reference images from the Wikipedia encyclopedia (Table 1).

Table 1: Exemplary color palettes comprising different amounts of colors used in this paper.

Palette Name	$\|P\|$	Palette Texture
Teletext / BBC Micro	8	
Apple II	15	
CGA	16	
Commodore 64	16	
Tandy	62	
Atari 800	122	

2.2 Minimum Color Distance Search

For matching two colors C_i and C_j, a distance function $d = \delta(C_i, C_j)$ is computed. We use the Euclidean distance in Lab color space [2] for the color distance computations.

Given an extracted color palette P, the mapping is basically a point-based image processing operation performing a linear minimum color distance search over all its entries for each pixel in the input image (R3, Equation (2), Algorithm 1).

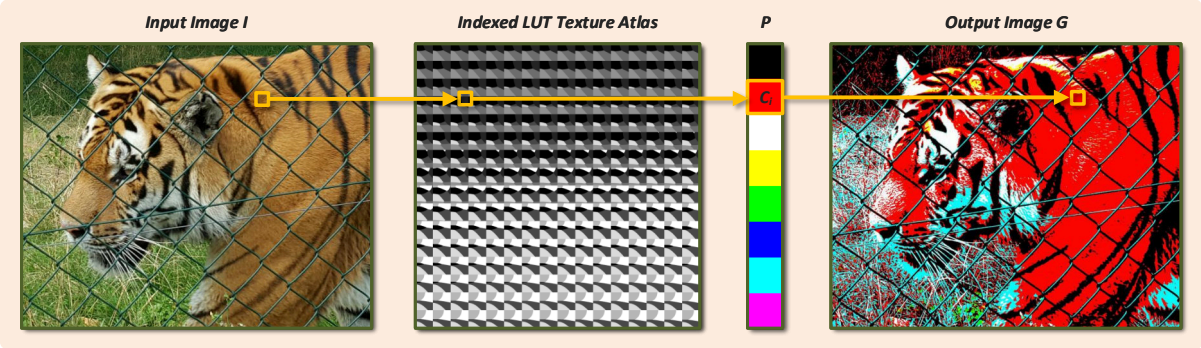


Figure 2: Color quantization using Indexed LUT (ILUT) using a BBC Micro 3 bit palette with 8 colors.

Algorithm 1 Minimum Color Distance Search

Require: $P = C_0, \ldots, C_n \; n > 0$ {Palette in Lab space}
  for $x = 0$ **to** w **do**
    for $y = 0$ **to** h **do**
      $d_{min} \leftarrow \infty +$ {Initialize}
      $C_{out} \leftarrow [0,0,0]$ {Initialize output color}
      $C_{in} \leftarrow I[x,y]$ {Fetch color value}
      $C_{in}^{Lab} \leftarrow rgb2lab(C_{in})$ {to CIELAB space}
      for $i = 0$ **to** n **do**
        $d = \delta(C_{in}^{Lab}, C_i)$ {Compute color distance}
        if $d < d_{min}$ **then**
          $d_{min} = d$ {Update min. difference}
          $C_{out} \leftarrow C_i$ {Update output color}
        end if
        if $d = 0$ **then**
          break{Early-out of search loop}
        end if
      end for
      $G[x,y] \leftarrow C_{out}$ {Set closest palette color}
    end for
  end for

2.3 Color Look-Up Table Generation

One observation of the minimum color distance search concerns its runtime complexity of $\mathcal{O}(whn)$ for an input image of width w, height h, and a color palette size of n. Considering the high spatial image resolutions of today's cameras, the algorithm's run-time performance easily exceeds the requirement of real-time performance (R1). Thus, the required number of searches is reduced to constant time by using LUTs covering the complete color domain. Similar to the approach described by Selan, a 3D color LUT can be computed in a pre-processing step on a per-palette level [5].

However, in contrast to this approach, we cannot take advantage of using bi-linear texture filtering to reduce the spatial resolution of the LUT. This would potentially introduce colors not represented by the initial palette and yield an imprecise mapping if using nearest-neighbor interpolation. One can distinguish between two LUT variants:

Color LUT (CLUT): This LUT type encodes the mapped palette color for each input color in Red-Green-Blue (RGB) color space (Figure 4(b) and Figure 5(a)). The consumed memory for this representation is $256^3 \cdot 3 \cdot B_C$ with B_C the number of bytes required to represent a color channel. For the palettes used in our applications it is sufficient to use 8 bit channel precision resulting in a Video Random Access Memory (VRAM) footprint of 50 MB

Indexed LUT (ILUT): Instead of storing the mapped color values directly (Equation (1)), the memory footprint can be reduced by storing only the respective color indexes (Figure 5(b)). Thus, the space-complexity reduces to $4096^2 \cdot B_I + |P| \cdot 3 \cdot B_C$, with B_I the number of bytes required per index. Usually, it is sufficient to use an 8 bit index precision, resulting in a VRAM memory footprint of 16 MB in addition to the color palette. Figure 2 shows an overview of the color mapping process using an ILUT.

3 GPU-BASED IMPLEMENTATION

The concept of the previous section is prototypical implemented using OpenGL [4] and OpenGL Shading Language (GLSL). However, these implementations can be easily transferred to other graphics Application Programming Interface (API) as well as Embedded System (ES) variants.

The remainder of this section briefly describes how palettes and different LUT types are represented on GPUs, and how the particular color mapping processes are implemented by the example of using OpenGL fragment shader programs. Such implementations are required because rendering using *paletted textures* are not supported by OpenGL and respective legacy formats are declared deprecated since OpenGL 3.0. Further, the support for the `GL_EXT_paletted_texture` extension has been dropped by the major hardware vendors.

3.1 GPU-based Palette Representation

Before describing the different palette mapping implementation, this section briefly present the palette and LUT representations suitable for GPU-based rendering.

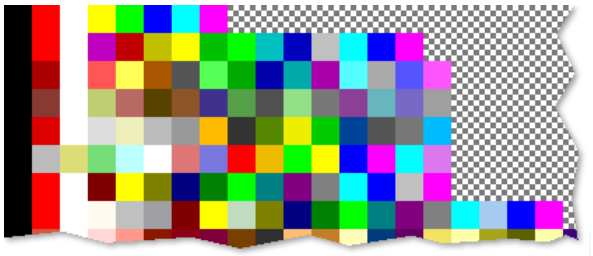


Figure 3: Close-up of an exemplary 2D TA that stores multiple color palettes row-wise.

3.1.1 *Texture-based Palette Representation*

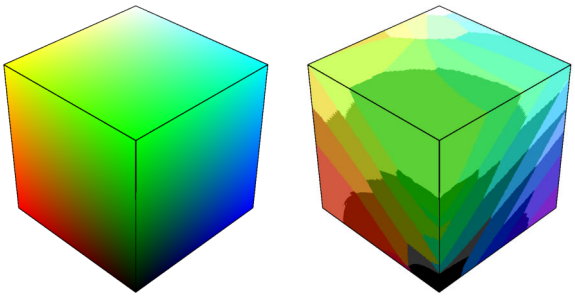
In general, each palette is represented using a single 2D texture. During runtime, a palette is converted into Lab color space [2] and uploaded to a *palette texture* within VRAM. The straightforward approach to standard palette representation for GPUs is using a three channel 1D texture, i.e., a 2D texture with fixed height of 1 pixel. Table 1 shows some texture examples. This approach provides optimal texture utilization.

However, the total amount of palette colors is limited to the maximum texture size. In rare cases that this size is exceeded, the palette is wrapped and indexing is performed by modulo operation. To minimize state changes during rendering, which can be introduced when switching palette textures, and to facilitate fast palette interchanges, such texture representation can be combined with texture batching [6]. This enables the storage of multiple palettes using a 2D Texture Atlas (TA) (Figure 3).

3.1.2 *Look-up Table Representations*

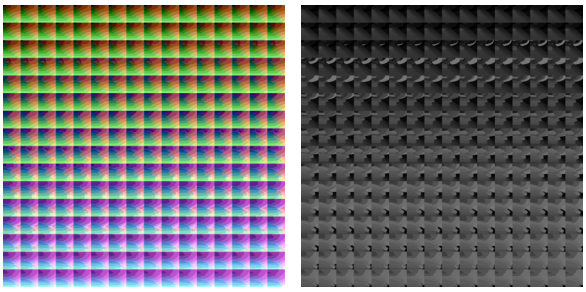
There are basically two texture-based representations of CLUTs and ILUTs for GPU-based implementation that take advantages of build-in texturing functionality of shading languages:

3D Textures: An effective way to represent a LUT is using a 3D texture of size 256^3 texels for precise mapping of low precision color representations (Figure 4). 3D textures are also supported for mobile devices since OpenGL ES 3.0 and thus are commonly available.



(a) *Identity Palette.* (b) *Quantized Palette.*

Figure 4: 3D texture LUTs.



(a) *CLUT TA.* (b) *ILUT TA.*

Figure 5: Comparison of 2D texture atlas LUT representations; each of 4096^2 pixels spatial resolution.

2D Texture-Atlases: Figure 5 shows a comparison between a TA containing all 256 layer of a CLUT (Figure 5(a)) and ILUT (Figure 5(b)). Such an atlas can be used in combination with 2D texture arrays to support fast interchanges of palettes (R2).

The texture format used for CLUTs is `GL_RGB` with an internal `GL_UNSIGNED_BYTE` representation is sufficient. For ILUT, `GL_LUMINANCE` using `GL_FLOAT` internal format at a minimum of 16 bit precision to represent the range of possible palette indexes for $n > 256$.

3.2 Shader-based Color Quantization

The shader-based color quantization can be performed within a single rendering pass. Therefor, a Screen-aligned Quad (SAQ) that is textured with the input image covering the complete viewport is rasterized with an activated fragment shader program and bound texture resources. The remainder of this section shows the particular fragment shaders for performing the color mapping operations for the respective texture representations described previously.

3.2.1 *Minimum Color Distance Search*

For texture-based palette representations, Listing 1 shows a GLSL implementation for the minimum color distance search (Section 2.1). Performed run-time tests comparing this implementation with variants that support early-out of the for loop (Algorithm 1) that, however, yield inferior performance presumably due to coherency issues [3].

3.2.2 *Sampling Look-Up Tables*

For sampling LUT textures, the input color values $C = (r,g,b) = (s,t,r) = I[x,y] \in [0,1]^3$ fetched at the respective fragment coordinates $(x,y) \in [0,0] \times [w,h]$ are used. While sampling 3D textures is natively supported by GLSL language features, LUTs that are represented by 2D TAs require additional instructions to resolve the indirection. Listing 2 shows a GLSL function for sampling LUTs represented as a 2D TA.

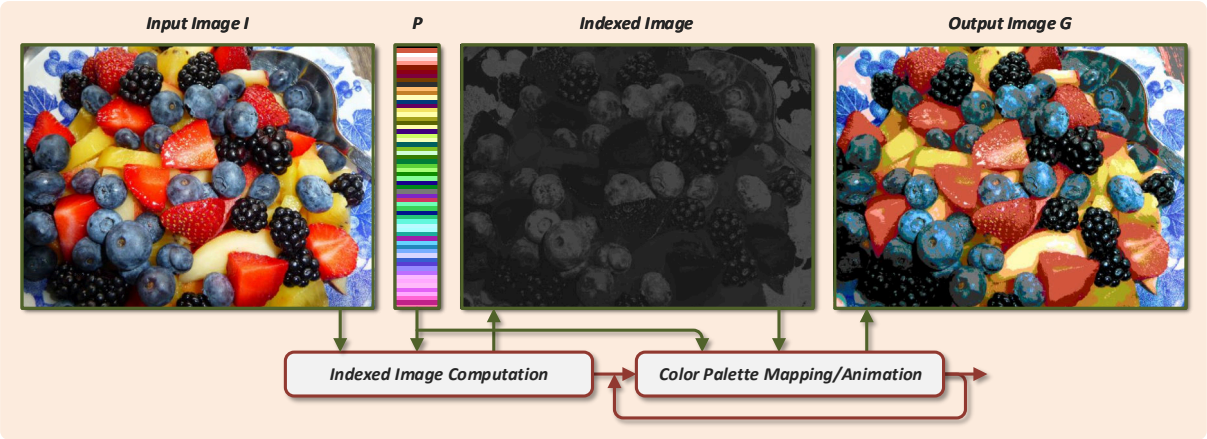


Figure 6: Overview of indexed image computation and subsequent color quantization by the example of an Atari 800 color palette with 122 colors.

```
int closestColorIndex(
  const in vec3 sampleLab, // Color in Lab
  const in sampler2D P,    // Palette TA in Lab
  const in int index)      // Palete to use
{
  // Determine palette length for palette index
  int pSize = texelFetch(P, ivec2(0, index), 0).x;
  int colorIndex = 0;
  float minimumDistance = FLT_MAX;
  // Linear search in palette
  for(int i = 1; i < pSize; i++) {
    // Compute color distance
    float d = distance(
      texelFetch(P, ivec2(i, index), 0).rgb,
        sampleLAB);
    // Minimum distance compare
    if(d < minimumDistance) {
      minimumDistance = d;
      colorIndex = i;
    }
  }
  return colorIndex; // Index of closest color.
}
```

Listing 1: GLSL implementation of minimum color distance search.

```
vec2 offset(const in float s, const in vec4 info){
  return info.zw * vec2(mod(s, info.y),
    floor(s * info.z));
}

vec4 sampleAs3DTexture(
  const in sampler2D LUT,
  const in vec3 texCoord,
  const in vec4 info) {

  float sliceZ = floor(texCoord.z * info.x);

  vec2 slice0Offset = offset(sliceZ, info);
  vec2 slicePixelSize = info.zw / info.x;
  vec2 uv = slicePixelSize * 0.5 + texCoord.xy *
            (info.zw - slicePixelSize);
  return texture2D(tex, slice0Offset + uv);
}
```

Listing 2: GLSL implementation for sampling a 2D texture atlas using given 3D texture coordinates.

3.2.3 *Quantization by Color Look-Up Tables*

Once, the CLUTs are generated, the actual quantization can be efficiently performed by sampling these using the RGB color values as texture coordinates similar to the approach described by Selan [5]. Listing 3 shows a GLSL function for color quantization using an CLUT requiring a single texture look-up.

```
vec4 mapColorCLUT(
  const in vec3 color,     // color to map
  const in sampler3D CLUT) // look-up table
{
  return texelFetch(CLUT, ivec3(color), 0);
}
```

Listing 3: GLSL implementation of CLUT mapping.

3.2.4 *Quantization by Indexed Look-Up Tables*

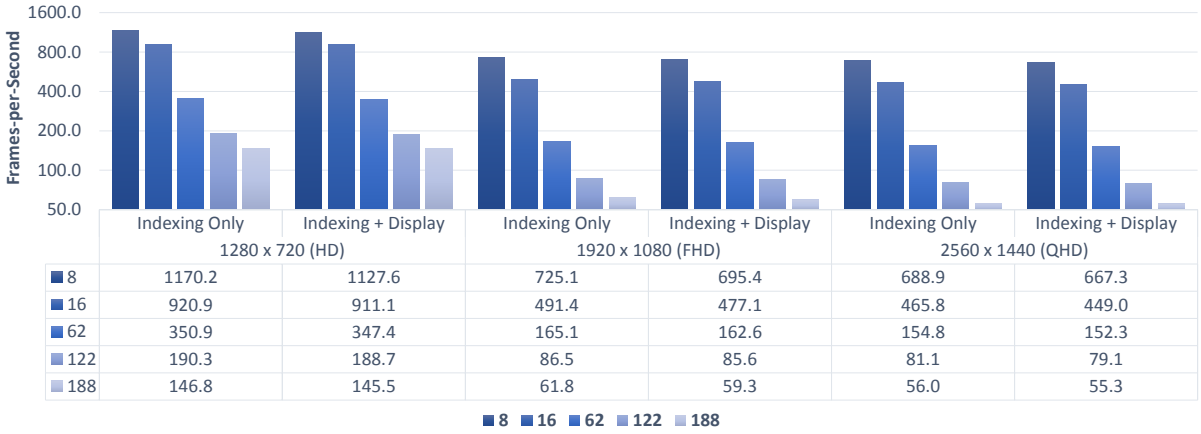
Compared to the CLUT-based quantization, ILUT-based quantization requires an additional sampling operation to resolve the indirection between ILUT and the color palette. Listing 4 shows a GLSL implementation for this mapping operation.

```
vec4 mapColorILUT(
  const in vec3 color,         // Color to map
  const in sampler2D palette, // Palette texture
  const in sampler3D ILUT)    // Indexed LUT
{
  // Obtain index into color palette
  int index = texelFetch(ILUT, ivec3(color), 0).r;
  // Sampling color palette
  return texelFetch(palette, ivec2(index, 0), 0);
}
```

Listing 4: GLSL implementation of ILUT mapping.

3.3 LUT and Indexed Image Generation

The task of LUT generation can be efficiently performed on GPU, for both 3D texture and 2D TAs using compute shader or off-screen rendering in combination with Frame-Buffer Object (FBO) and



	1280 x 720 (HD)		1920 x 1080 (FHD)		2560 x 1440 (QHD)	
	Indexing Only	Indexing + Display	Indexing Only	Indexing + Display	Indexing Only	Indexing + Display
8	1170.2	1127.6	725.1	695.4	688.9	667.3
16	920.9	911.1	491.4	477.1	465.8	449.0
62	350.9	347.4	165.1	162.6	154.8	152.3
122	190.3	188.7	86.5	85.6	81.1	79.1
188	146.8	145.5	61.8	59.3	56.0	55.3

Figure 7: Performance comparison of GPU-based LUT generation (indexing) and combined mapping (display) in Frames-per-Second (FPS) (logarithmic scale) for color palettes of different complexity (8 to 122) and different input image resolutions.

fragment shader support (can be optimized by `GL_EXT_shader_framebuffer_fetch`). Both are based on the minimum distance search implementation described in Section 3.2.1. In general, these techniques can be used to "bake" any kind of LUT. The overall goal is avoid color space conversion at mapping time by using RGB color space values only.

The 2D TAs processing approach can be used the generate an intermediate *indexed image* representation, that can be used for rendering animated palettes for mimicking *palette cycling*. Therefore, the input image I is transformed once to an indexed image G using the implementation of Section 3.2.1. Palette animations can be easily achieved algorithmically or represented using a 2D TA of palette texture key-frames (Figure 6).

4 PERFORMANCE EVALUATION

This section presents performance evaluations of all three approaches described before. In addition thereto, the run-time performance for color mapping and the respective memory footprints of the particular data representations are compared.

4.1 Datasets and Test Systems

Different image resolutions were tested to estimate the run-time performance regarding the spatial resolution of an image as well as different palette complexity (Table 1). The following common resolutions were chosen: 1280×720 (HD), 1920×1080 (FHD), and 2560×1440 (QHD) pixels. We tested the rendering performance of our preliminary implementation was conducted using a NVIDIA GeForce GTX 970 GPU with 4 096 MB VRAM on a Intel Xeon CPU with 2.8 GHz and 12 GB RAM. Rendering was performed in windowed mode with vertical synchronization turned off.

The measurements in FPS are obtained by averaging 500 consecutive frames. For all off-screen rendering passes, fragment shader functionality using a textured SAQ with a geometric complexity of four vertices and two triangle primitives. For rasterization, back-face culling [1] is enabled and depth test and writing to depth buffer are disabled.

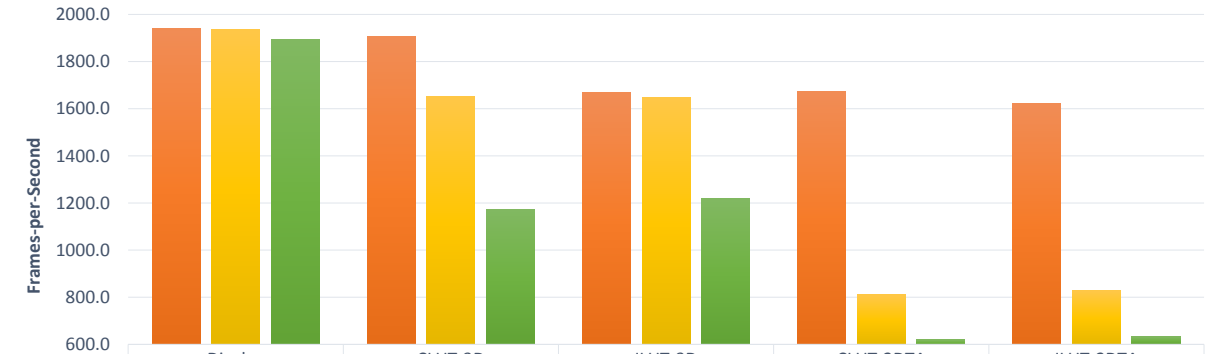
4.2 Performance Results

Figure 7 shows the performance evaluation for the minimum color distance search implementation (Section 3.2.1) for different color palette complexities (8 to 188 colors) by comparing the computation of an indexed image representation G only (*indexing*) as well as in combination with a subsequent display pass that applies the mapping implementation of Section 3.3.

It can be observed, that run-time performance of indexing decreases linearly with increasing color palette complexity, while the performance impact of color mapping during display is neglectable. The overall performance decrease with respect to increasing input image resolution indicates that the technique is fill-limited.

Figure 8 shows the results of the run-time performance evaluation in FPS of the different GPU-based LUT representations (CLUT and ILUT using 3D textures and Texture Atlass (TAs)), algorithms (Section 3.2.3, and Section 3.2.4) impacting the color mapping performance compared to a pass-through display pass.

It can be observed that 3D texture representations for CLUT and ILUT are superior over 2D TAs, especially for high spatial input resolutions. This can be explained by the additional instructions required to transform the texture coordinates to sample 2D TAs and probable texture-cache violations. Independent of their representations are ILUTs inferior to CLUTs. This can be explained by the additional texture sampling operation required to resolve the palette indirection, which most probably contributes to additional texture-cache viola-



	Display	CLUT-3D	ILUT-3D	CLUT-2DTA	ILUT-2DTA
1280 x 720	1938.5	1907.0	1667.9	1673.3	1623.7
1920 x 1080	1935.7	1651.6	1649.1	810.3	828.3
2560 x 1440	1893.3	1174.2	1218.5	621.2	633.9

Figure 8: Performance of GPU-based LUT mapping performance in FPS for different LUT representations compared to a pass-through display pass.

tions. These observations show all characteristics of the traditional space-time/time-memory trade-off.

5 CONCLUSIONS

This paper presents and discusses an approach for fast real-time color palette quantization for true-color input images. The palette quantization is based on computing minimal color difference using the Lab color space. It is shown, that this mapping can be performed in real-time and can be significantly optimized by using LUTs computed in a pre-processing step on GPUs. Depending on the use-case, the described approaches allows for trading run-time performance for VRAM memory consumption.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work was funded by the Federal Ministry of Education and Research (BMBF), Germany, for the AVA project 01IS15041.

REFERENCES

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018.

[2] B. Hill, Th. Roger, and F. W. Vorhagen. Comparative analysis of the quantization of color spaces on the basis of the cielab color-difference formula. *ACM Trans. Graph.*, 16(2):109–154, April 1997.

[3] Marc Olano, Kurt Akeley, John C. Hart, Wolfgang Heidrich, Michael McCool, Jason L. Mitchell, and Randi Rost. Real-time shading. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[4] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification Version 4.5*. Silicon Graphics Inc., 4.5 edition, June 2017.

[5] Jeremy Selan. Using Lookup Tables to Accelerate Color Transformations. In *GPU Gems*, pages 381–392. Addison-Wesley, 2004.

[6] Matthias Wloka. *ShaderX3*, chapter Improved Batching via Texture Atlases, pages 155–167. Charles River Media, 2005.